Program Specification Applied to a Text Formatter

MARTIN S. FEATHER

Abstract—Presentation of the formal specification of a small text formatter illustrates an approach to the construction of formal specifications. The key features of this approach are described, and their beneficial influence on the construction and organization of specifications of tasks, especially those for which no concise descriptions are possible, are discussed. The intent is that in addition to serving as formal descriptions of tasks, such specifications will be of use in the processes of verification, development, and maintenance of their implementations.

Index Terms—Applicative programming, program reliability, program specification, programming techniques, program transformation.

I. Introduction

A SPECIFICATION differs from an implementation in that there is no need for it to be efficient in the computational sense; rather, our sole aim in constructing a specification should be to maximize our confidence that the specification in fact denotes the behavior we desire. Within this paper we will be concerned with activities for which no concise specification is available—in such cases we must expend effort to develop a well-organized specification so that despite its size it is nevertheless comprehensible.

Our interest in specification derives from work on program transformation, in particular transformation based upon the methods for manipulating recursion equations, as developed by Burstall and Darlington [9]. From this work Burstall was motivated to create a simple recursion-equation programming language, NPL [8]. We have used NPL as the language in which to express our specifications, and will consider how its use influenced our construction of specifications. NPL has since been rationalized and extended to become a more powerful language, HOPE [10]. The work to be described here was done during the lifetime of NPL; at the end we will comment briefly on how HOPE's additional features might have been of further help.

We consider one particular task, and show how, by using the recursion-equation language and freeing ourselves from all consideration of efficiency during design, we may emerge with a good formal specification. The domain of the task is text formatting; we limit our attention to the simpler end of this domain, and adopt the set of features of an already defined formatter as those we must specify. The formatter we adopt is that described by Kernighan and Plauger [20, ch. 5]. In this book the authors demonstrate, with the aid of examples, how to organize one's approach to programming to go from an informal task description to a reasonably efficient and well-organized program to perform that task. Hence they provide both an informal description of a text formatter, and a program to do the formatting (written in Ratfor, i.e., preprocessed Fortran). We will construct a formal specification of their formatting task and investigate the benefits we claim for such a formal description.

Manuscript received February 23, 1981; revised July 23, 1981. This work was supported by the Science Research Council and was performed within the Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.

The author is with the Information Sciences Institute, University of Southern California, Marina del Rey, CA 90291.

II. Constructing Specifications

We see two main properties that a formal specification should have.

1) It must in fact denote the behavior the writer desires.

2) It must serve as a clear description which can be read and comprehended.

In the case of problems for which no concise specification is apparent it becomes much more difficult to achieve both of these goals. We feel that we have benefited from using NPL in which to express our specifications, insofar as it has encouraged a clear organization of our specifications. The main features of the language that we feel have provided this benefit are as follows.

1) The applicative nature of the language leads to a style of programming which is clearer and less error-prone. Destructive operations, side effects, and iteration are mechanisms appropriate to achieving efficiency but reduce clarity, since their inclusion renders communication between portions of the specification much less transparent. A crucial need in the successful organization of a large specification is to decompose it into separate components, each of which may be understood in isolation, and the communication between which is straightforward enough to permit comprehension of their combination.

2) The language is strongly typed and permits user defined types. By making liberal use of types defined for the task being specified we get support from the type-checker and provide additional helpful information to the reader.

3) Writing in recursion equations encourages decomposition of the overall large problem into simpler subproblems, which in turn may be decomposed, until finally we emerge with many trivial problems each of which may be easily coded.

In the example to follow we will see the above influences in action. The presence of these features is, of course, no guarantee that we will emerge with a suitable specification. We must adopt a style which makes good use of them. To this end, the freedom to disregard efficiency is of crucial importance. This underlies the crucial difference between our approach and that of structured programming. Whereas the latter is a means for developing a tolerably efficient program, we are concerned (at this stage) solely with developing a specification. Hence, although we may indeed adopt some of the organizational techniques of structured programming, we will typically choose to decompose a problem in a radically different manner, one better suited to satisfying the needs of a specification.

We emphasize that no elements of our approach are new—rather, we are following the techniques already suggested by other researchers. Backus [2] has argued for the need to escape from conventional imperative programming; Burge [7] investigates recursive programming. The typed nature of the language ML [19] influenced NPL. Noted texts on structured programming include [12] and [16]. Balzer *et al.* [3], Bauer *et al.* [4], and Darlington and Burstall [14] advocate developing programs by first constructing a specification and then transforming to introduce efficiency.

III. The Example Task

The example task we consider is a (small) text formatter. This we chose as a reasonably well-understood task, one of sufficient complexity that no concise specification is possible (at least, none that we know of), hence suitable as a problem for trying the method of developing a program by formal specification followed by transformation.

A. Informal Description of a Text Formatter

We give a (very) brief and informal account of the facilities the text formatter is to provide.

Input to the formatter is a sequence of lines, where lines consist of sequences of characters. Some lines will be text, some will be commands to the formatter. Command lines are identified by the occurrence of a "." in the first column followed by a two letter abbreviation of the name of the command.

In action the formatter may be in a "fill" mode, during which paragraphs are formed by packing as many input words as possible into the output lines, the lines being "right-justified" (to produce an aligned right margin, like this paragraph) by padding out with extra spaces between words if necessary. When not in "fill" mode the input text lines are output without modification. When switching off filling, the words already gathered to go into the next output line are put out without right justification. This action of forcing out a partially collected line is called a break. Some of the commands implicitly cause breaks when they are encountered, even though they may not cause filling to be switched off.

We present the commands and briefly explain their actions:

"Filling" commands

fi Cause a break and switch on "fill" mode.

nf Switch off "fill" mode.

br Cause a break (but does not switch into or out of of "fill" mode).

Page commands

bp n Begin page. n is an optional numeric argument, which, if present, is taken as the number of the new page. If not present the default is to increment the current page number by one. Causes a break. If this command would produce an entirely blank page (but for header and footer titles), i.e., occurs at the very top of a page, it merely adjusts the page number without creating the blank page.

pl n Set page length to be n lines. Default is n=66, does not cause break.

he t Set the header to be printed at top of each page. t is a string argument which becomes the new header. The character "#" within the string is replaced by the current page number. Does not cause a break.

fo t Set the footer title to be printed at bottom of each page. Analogous to **he** command.

ls n Set line spacing to n (i.e., n=2 corresponds to double spacing). Default is n=1, does not cause a break.

sp n Causes a break and produces n blank lines. Default is n=1. Does not produce blank lines at the very top of a page.

Line commands

ce n Cause a break and center the next n text lines (i.e., insert extra spaces if necessary to cause the text lines to be centered within the current margins.) Default is n=1. If another **ce** command is encountered whilst centering text lines, the new command's value of n takes precedence.

ul n Does not cause a break. Default is n=1. As with ce command, encountering another **ul** command will adjust the count of lines to be underlined.

rm n Set right margin to be n. Default is n=60, does not cause a break.

in n Set left margin (indentation) to be n. Default is n=0, does not cause a break.

ti n Cause a break and set the left margin for next output line only to be n. Default is n=0.

Numeric arguments to commands may be preceded by a "+" or "-", in which case the value is taken to be the current value of the parameter being set incremented or decremented accordingly. An exception to this is the **ti** command which adjusts relative to the current left margin setting.

In order that the formatter behave reasonably with text containing a minimum of formatting commands, input lines which start with blanks or are entirely blank are treated as follows.

Lines empty but for blanks cause a break and a blank line to be output (even at the top of a new page).

Lines starting with n blanks (but followed by other characters) where n>0 cause a break and a temporary indent of +n.

B. The Organization of our Specification

The formatting task may be characterized as follows: from the input representation we extract the lines of text and the associated information which will direct the layout of that text. These lines are processed as directed to produce output lines representing pages containing paragraphs, verbatim text, headers, etc.

Our specification will follow this characterization, i.e., we will have a first stage in which the input is decoded to extract the text lines and associate with each the information to direct the formatting; and a second stage in which these lines + information are processed into paragraphs, pages, etc. from which output lines can be formed.

Thus, already we see a major divergence between our specification and any reasonably efficient program—we have two distinct stages connected by the passing of a bulky but conceptually simple data structure, whereas an efficient program would perform the whole operation in a single pass, incrementally maintaining the current information and producing output.

Now we tackle the decomposition of these stages.

Decoding Input to Associate Information with (Text) Lines: This breaks down into two more stages; the first to recognize command lines and decode the type of command and arguments (if any). The output of this is a sequence of elements, each of which is either a (text) line (i.e., sequence of characters) or a command (some types of commands having argument values associated with them). Fig. 1 displays the overall organization of our specification, and within it we call this stage of the processing DECODE.

The second stage associates with each text line a data structure to hold the information relevant to formatting that line, e.g., margin values, page size, etc. We call this collection of information an "infomap." Thus output from this, to the second main stage of the whole formatting process is a sequence of elements, each of which is a text-line + infomap. Actually this is a slight oversimplification, insofar as we leave commands of a few types within this sequence rather than trying to force the information implied in the commands into infomaps. The aberrant commands are:

sp (space down)—left untouched because it is only during page formation that we may determine whether the blank lines this generates would fall at the very top of a page (in which case they are to be discarded), or whether they would fit into the remaining space on the current page (if not they fill it to the bottom, but do *not* overflow onto the next page).

bp (begin page)—left untouched because it is only during page formation that we may determine the page number of the current page, and this command might specify a relative change to the page number rather than absolute.

br (break)—the sole purpose of this command is to delimit collection of words to be accumulated into a single paragraph.

We call this stage DOCOMMANDS within Fig. 1. This decomposes further into a separate stage for each command wherever possible. The **fi** and **nf** commands (to switch filling on and off) interact in such a way as to necessitate handling together, as do the **in** and **ti** commands (indent and temporary indent). Apart from these pairs the commands are dealt with in separate passes, the order of which is irrelevant.

Processing of Text Lines—This breaks down into four distinct stages.

1) INTERMEDIATE: Do processing local to individual text lines, i.e., center those lines which need centering by adjusting the margin values within their infomaps, follow each character in text lines whose contents are to be underlined by backspace and underline, and deal appropriately with text lines which start with blanks and/or are entirely blank. Each of these activities is done in a separate substage—they have been grouped together here because they refer to information and make changes purely within single text lines.

2) LINES: Within this stage paragraph formation is performed and overlength lines are dealt with (resulting in a split into two or more lines). The distinction between this and the INTERMEDIATE stage is that this involves actions spread over possibly several text lines. Paragraph formation is the most interesting of the activities done in this stage. We break this down into

• gathering the lines from which words to go into a single paragraph are to be extracted;

• extract the words from these lines (and associate with each word the "infomap" of the line from which it has been extracted);

• squeeze as many of the words as possible into each successive line to form a filled paragraph, and perform justification on each such line. In forming each line we let the infomap of the first word to go onto the line determine the characteristics of the entire line (in particular, its margin sizes).

3) PAGES: The text lines to go into pages have been formed in the previous stage, and within this stage these lines are not modified in any way, merely accumulated into pages. Again, a breakdown into subtasks is followed to accumulate page-filling sequences of lines and form actual pages from these.

4) OUTPUT: Finally, the sequence of pages is simplified into a sequence of lines for output to some printing device. It is here that we would tailor the output to whatever device was to be the destination (e.g., if the device had no backspace character but could overprint an entire line, we would modify those lines with backspace accordingly.)

We could provide further detail about each stage of processing, however we feel that we have sufficiently demonstrated the overall approach to decomposition of the problem. A portion of the NPL code which forms our specification is provided in the Appendix.

C. Implications of Having Constructed a Specification

We consider what benefits there may be from having a formal specification, and reflect on what the exercise has revealed about formatting the specification in general.

1) Increased Understanding of Task: One consequence of our attempt to produce as clear a specification as possible is that it lead us to consider some aspects of formatting that we might otherwise have overlooked.

An example of this arose during specification of paragraphing, when words are extracted from the incoming text lines and put into filled and justified lines. It is clear that the information associated with an input text line should become the information associated with each of the words extracted from that line. However, it is not so trivial to decide how the information associated with several words will be used to determine the information to be associated with the output text line that they are to form. In our specification we chose to let the information associated with the first word to go onto a line determine the information for the entire line.

```
lines (each a sequence of characters)
  ↓  DECODE–recognizes command lines and decodes them
(text) lines interspersed with commands
  ↓  DOCOMMANDS–associates with each text line on "infomap"
                 to hold formatting information values
(text) lines + infomaps interspersed with remaining commands
  ↓  INTERMEDIATE–local line manipulation (underlining, etc.)
(text) lines + infomaps interspersed with remaining commands
  ↓  LINES–form output lines (involving filling and justifying
            paragraphs, centering lines between margins, etc.)
(text) lines + infomaps interspersed with remaining commands
  ↓  PAGES–form output pages (generating header and footer
            titles, padding with blank lines, etc.)
pages (each a sequence of (text) lines)
  ↓  OUTPUT–simplify to a sequence of lines
lines
```

Fig. 1. Structure of specification.

This is *not* the behavior of Kernighan and Plauger's efficient formatter. Their program, when accumulating the words to go into a filled line, is simultaneously taking notice of the incoming commands to change formatting values (in particular, right margin value). Hence, in the middle of forming a filled line the right margin value might change, and the line formed would reflect the latest value rather than the value prevalent at the start of the line. We speculate that this is an implicit consequence of their aim to design an efficient program rather than the result of a conscious choice between alternatives. Their algorithm may exhibit unusual behavior if during accumulation of words to go into a line the right margin should decrease to less than that needed to accommodate the words accumulated so far—the effect is to cause those words to be put out in a line whose rightmost margin will be neither the old right margin value nor the new smaller value. For example,

```
                              Our Specification's    Kernighan & Plauger's
        Input                       Output             Program's Output

.rm 14                        AA BB CC DD EE         AA BB CC DD EE
.fi                           FF GG HH II JJ         FF GG HH II JJ
AA BB CC DD EE FF             KK LL MM NN OO         KK LL MM
GG HH II JJ KK LL MM          PP QQ                  NN OO
.rm 5                         RR SS                  PP QQ
NN OO PP QQ RR SS                                    RR SS
```

Our point is not that our specification exhibits the "right" behavior, but rather that in its construction we were led to consideration of the options available. Regardless of whether we make any further use of our specifications, we would claim that, by being led to such considerations, we benefit in gaining a better understanding of the task in question.

Some further observations that we are led to make are as follows.

Page Formation: There is a close parallel between filling lines with words and filling pages with lines. In each case we must decide what effect commands which adjust line/page parameters should have when they occur in the middle of forming a line/page. (Of course, commands that cause a new line/page to be started have a clear intent—it is only those which adjust parameters without terminating the current line/page whose precise effect is unobvious).

In the same way that we must decide how the information associated with the words to go into a line should determine the information for that line, we also must decide how the information associated with the lines to go into a page should determine the information for that page. To be consistent we choose to let the information of the first line serve as the information for the entire page.

Oddities of Line Filling and Page Formation: Some of the consequences of permitting parameter changes to occur without forcing line/page breaks are rather strange. For example,

```
        Input                   Output

.rm 14                    AA BB CC DD EE
.fi                       FF GG HH II JJ
AA BB CC DD EE FF         KK LL MM NN OO
GG HH II JJ KK LL         PP QQ RR SS TT
.rm 7                     UU VV WW XX YY
MM NN                     ZZ
.rm 14
OO PP QQ RR SS TT
UU VV WW XX YY ZZ
```

We might be surprised to observe no lines with right margin set to 7 in the output. This is a consequence of our decision to let a line's first word's information set the characteristics for the entire line—here the decreased right margin had no effect on the line already in progress, and by the time the next line begins, the margin has been readjusted!

We might also be surprised to learn that the obvious policy of filling up the current line with as many words as possible before starting the next line does not necessarily lead to the paragraph of the shortest overall length! For example,

```
Input              Our Output          Alternative

.rm 15
AAA BBB CCC DDD    AAA BBB CCC DDD     AAA    BBB    CCC
.rm 8              EE FF GG            DDD EE FF GG HH
EE FF GG HH        HH      III         III   JJJ
III   JJJ          JJJ
```

Here by allowing word "DDD" to spill over onto a new (and hence longer) line, more of the following words can be accommodated before the margin shrinkage comes into effect, resulting in a decrease in the paragraph length.

These behaviors suggest that we should consider revising our definition of the formatting process to insist that commands modifying parameters of a line/page always have the effect of terminating the current line/page.

2) Availability of a Clear Specification: We have more confidence that we understand the behavior implied by our specification than that of Kernighan and Plauger's efficient program. This may be due in part to our having performed the design of our specification, but served only as a reader of their program. However, a good deal of our confidence may be attributed to the deliberately simple structure of our specification, from which we are better able to perceive how the portions interact, and the likelihood of there being anomalous behavior in circumstances that we had not anticipated much reduced.

3) Areas of Dissatisfaction: We are reasonably happy about the overall quality of our resulting specification; however, we perceive some areas in which we harbor some lingering dissatisfactions.

With respect to clarity, we were forced to make compromises between complicating some portions of the specification in order to simplify others. An example of this is to be found in the method by which information to control line and page formation is gathered and disseminated. Ideally, at any stage which makes use of some of the information it should be evident what information is necessary for the activities of that stage. We have simplified the gathering and passing of information between stages so that each stage receives text lines together with so called "infomaps," containing all the associated information. Hence, it is only through inspection of the activities that we may determine which portions of information are actually required and which are not (e.g., to determine that the pages formation stage makes use of page-length information, but not margin sizes). An alternative would have been to complicate the stage that gathers "infomaps" in order to partition the information so that each stage may be given only the portion of the information necessary for its activities.

Another example is our decision to leave some of the commands (**br**, **bp**, and **sp**) interspersed with the (text) lines, rather than go to the extra effort of encoding them into infomaps. Again, we see this as a compromise rather than a clear-cut best choice.

4) Availability of a Precise Specification: Our specification is formal as opposed to informal; hence we may use it in the following ways.

Testing: The NPL interpreter may be used to run the specification on small examples to test its overall behavior. The inefficient nature of the specification prohibits extensive testing; however, its design, in which the overall task is decomposed into smaller and simpler subtasks, permits extensive testing of the components. In practice, our mode of construction and testing of NPL specifications goes as follows: first, we design a specification in the manner outlined for the test formatter; second, we turn this design into an NPL program, which we feed into the type checker, at which point syntactic errors are discovered and removed; third, we are left with a syntactically correct program, which we may try small test cases on, and the components of which we may test extensively. We find that the few errors present in the program are glaringly obvious and relatively simple in nature (both to understand and correct).

Verification: We may attempt to prove properties of our specification. The applicative nature of NPL and the simplicity of our specification make proofs easier. Indeed, Burstall and Darlington acknowledge the influence of Boyer and Moore's successful theorem prover for properties of Lisp programs (see [5] and [6]) on their decision to investigate the transformation of recursion equations. Although we do not have a theorem prover for NPL (or HOPE) programs, the domain of Aubin's prover ([1]) is a language of recursion equations very similar to NPL, and would seem to be most appropriate.

Transformation: At the start we remarked on how NPL originated from Burstall and Darlington's transformation work. Our intent in producing specifications in NPL is to then transform them into more efficient NPL programs. Darlington has created a transformation system for NPL, described in [13]. His system is more geared to researching the extent to which transformation techniques may be automated than to performing large transformations which, in the present state-of-the-art, require significant user guidance to be practical. We have engineered a transformation system for NPL which is designed primarily to support the human user in performing large transformations—details of this system may be found in [18]. With this we were able to transform our text-formatter specification into an NPL program with the structure and behavior of a more conventional algorithm (i.e., a one-pass algorithm in which formatted output is produced as the input is consumed, with the consequent vast improvement in efficiency over our specification, at the expense of clarity). All the steps of this transformation process were machine applied, hence we have confidence that the result of the transformation is in fact equivalent to our specification. Furthermore, the record of human direction to the transformation system is preserved as a manipulable structure in its own right—one which may be read, reapplied, modified, etc.

Maintenance: We would like to see the specification, transformed efficient program, and record of transformation serve together to support maintenance. Rather than attempting to modify the efficient program directly—an error-prone activity at best—we would perform our modifications upon the specification (which, because of its very nature should admit to easier and more error-free maintenance) and reperform the transformation to produce a modified efficient program. Should the nature of the modification be concerned solely with adjusting the efficiency rather than the functional behavior of the program, we would adjust the transformation while leaving the specification the same. These are, at present, wishes only. A good deal of research needs to be done into the maintenance aspect of transformation. See [15] for a report of preliminary experiments in this direction.

Alternative Evaluation: The NPL interpreter evaluates NPL programs in a straightforward call-by-value fashion. However, the applicative nature of recursion equations makes them neutral to the order of evaluation (except perhaps for loss of termination). Schwarz [22] investigates how we may augment an NPL program with control information to direct a more sophisticated interpreter in its selection of evaluation mechanism. By this means we may make some improvement to the efficiency of the interpretation without resorting to transformation; see also [21] and [11] for related work on the language Prolog. Darlington has developed some ideas on how to evaluate recursion equations on parallel hardware, and has observed that the specification style of lavish decomposition is suited to taking full advantage of parallel processing power.

IV. Conclusions

A. Accomplishments

We have constructed a specification of a simple text formatter, and would like to think that our specification has the virtues of clarity, comprehensibility, and precision. We ascribe the success we might have had to the influences inherent from using an applicative recursion equation language, and to the deliberate decision to disregard all consideration of efficiency. We feel that we have benefited from the experience both by emerging with a formal specification, and by having gained some insights into the task of formatting itself. Further details of our efforts, both specification and transformation, may be found in [17].

B. From NPL to HOPE

As we remarked earlier, the language NPL has since been developed further to become HOPE; the two main extensions, and how they might have been of use to us, are as follows.

Higher Order: NPL is a first-order language, without the power of passing functions as values to functions. We do not think the availability of this feature would have influenced our design of the specification, merely simplified expression of some of the low-level activities.

Modularity: NPL contains a very crude mechanism for information hiding. Within HOPE this mechanism has been significantly extended into a modularization facility. We might make good use of this feature to make explicit the separation of the various stages of our specification. We do not think this would have influenced us to adopting a very different design, rather encouraged us (by the support it provides) to make clearer the communication of information within the established configuration.

C. Avenues for Further Research

Restricting our attention to the specification aspect, we see the following possibilities for investigation.

Modularity: We speculate that the apparent compromise between simplicity of some portions of the specification and explicating the necessary and unnecessary communication between portions signifies a weakness in the modularization of our program. We require experience with facilities such as those incorporated into HOPE to determine whether this problem can be partially or wholly overcome by language support.

Explanation: Although we are reasonably happy with our NPL specification, we recognize that it will not suffice as a self-explanatory document; we ascribe its failing in this respect to two factors: first, it is written without redundant information, and makes no attempt to build up from anew to an understanding of the whole; second, the decisions made during construction and the reasons for making those decisions are not evident in the final product of the construction process. There is a need for investigation into providing such support for specifications.

Appendix
Portions of NPL Specification of Text Formatter

Portions of the NPL code forming our specification are shown. In order to make these comprehensible to the reader, we first describe NPL notation, illustrated with some trivial examples.

Introduction to NPL Notation

NPL programs consist of data declarations and function definitions.

A data declaration is used to introduce a new data type along with the data constructors which create elements of that type. For example,

<u>data</u> *num* = 0, succ(*num*)

defines a data type called *num* (to represent natural numbers) with data constructors 0 and succ. So the elements of *num* are 0, succ(0), succ(succ(0)), · · · .

A function definition consists of a type declaration and a sequence of one or more equations, where each equation specifies the function over some subset of the possible argument values. For example,

```
type declaration
   Factorial : num → num
equations
   Factorial(0) ⇐ succ(0)
   Factorial( succ(N) ) ⇐ ( succ(N) ) * Factorial(N)
```

defines the usual factorial function.

The type declaration indicates that **Factorial** takes a single argument of type *num*, and produces a result of type *num*.

The two equations specify **Factorial**; each is of the form pattern ⇐ expression. To evaluate an expression, the equations are applied as rewrite rules; the patterns of the equations are matched against portions of the expression (in a leftmost innermost first order), and a portion successfully matched by an equation's pattern is replaced by the correspondingly instantiated expression of that equation. For example,

Factorial(succ(0))	applying the second equation of **Factorial**, evaluates to
(succ(0)) * **Factorial**(0)	and applying the first equation, to
(succ(0)) * succ(0)	etc.

Function symbols may be used as prefix or infix operators, as is "*" (multiplication) in the above.

The data type list may be defined as follows:

<u>data</u> *list alpha* = nil, *alpha* :: *list alpha*

using the infix symbol "::" as the constructor (to CONS together lists), e.g., 1::(2::(3::nil)).

We use the following abbreviation for lists: abbreviate e_1::(e_2:: · · · ::nil) as [e_1, e_2, · · ·]. Thus [ ] = nil. [1] = 1:: nil, [6, 5, 4] = 6::(5::(4::nil)), etc.

Hence the conventional functions **Append** and **Reverse** on lists may be defined by

```
Append : list alpha x list alpha → list alpha
Append( [ ], L2 ) ⇐ L2
Append( A::L1, L2 ) ⇐ A::(Append( L1, L2 ))

Reverse : list alpha → list alpha
Reverse( [ ] ) ⇐ [ ]
Reverse( A::L ) ⇐ Append(Reverse(L), [A])
```

We commonly use the symbol "<>" as an infix form of **Append**, so L1<>L2 = **Append**(L1, L2).

Font conventions—to enhance readability, we use differing fonts as follows:

boldface for function names (e.g., **Factorial**),
italics for type names (e.g., *num*),
roman for constructor names (e.g., succ),
SMALL CAPITALS for variable names in equations (e.g., N).

Portions from the DOCOMMANDS Stage of Processing

The DOCOMMANDS stage associates with each text line a data structure to hold the information relevant to formatting that line. The data types we are dealing with include:

- *command* defined by <u>data</u> *command* = br, fi, nf, etc.
- *argument* defined by <u>data</u> *argument* = unsigned(*num*), signed(*sign*,*num*), string(*list character*), null.
- *text-or-command-line* defined by <u>data</u> *text-or-command-line* = text(*list character*), cmd(*command*,*argument*).
- *infomap* to hold a "map" of the information relevant to formatting a line, further details omitted here.
- *itext-or-command-line* defined by <u>data</u> *itext-or-command-line* = itext(*infomap*,*list character*), cmd(*command*,*argument*).

The first activity of the DOCOMMANDS stage is to associate with each text-line an initially empty infomap, done by function **InitializeInfomaps**.

```
InitializeInfomaps : list text-or-command-line → list
                        itext-or-command-line
InitializeInfomaps(L) ⇐ AddMap( empty-map , L )

AddMap : infomap x list text-or-command-line → list
            itext-or-command-line
AddMap( MAP , [ ] ) ⇐ [ ]
AddMap( MAP , text(CHARACTERS) :: L )
      ⇐ itext( MAP , CHARACTERS ) :: AddMap( MAP ,L )
AddMap( MAP , cmd(C,A) :: L)
      ⇐ cmd(C,A) :: AddMap( MAP , L)
```

The first equation of **AddMap** corresponds to the trivial case when its input is exhausted.

The second equation corresponds to the first element of the list of text-or-command-lines being a text-line, in which event the infomap is associated with the characters of that line.

The last equation corresponds to the first element of the list of text-or-command-lines being a command-line, in which event it is passed through unchanged.

Thereafter there are separate passes for each type of command, going through the list of itext-or-command-lines and adding the necessary information to the infomaps.

For example, processing of the ce (center) commands will result in each itext-line's infomap being augmented with true or false to denote whether or not that line is to be centered in the output.

Function **DoCentre** does this processing pass, using a subsidiary function **SubDoCentre**, and an initializing constant **InitCentreCount** to set the default number of lines we want centered at the start of the document.

```
DoCentre : list itext-or-command-line → list
              itext-or-command-line
DoCentre(L) ⇐ SubDoCentre(L, InitCentreCount)

SubDoCentre : list itext-or-command-line x num → list
                 itext-or-command-line
```

```
SubDoCentre([ ], COUNT) ⇐ [ ]
SubDoCentre(itext(IMAP, CHARS) :: L, 0)
        ⇐ itext(AddToMap(IMAP, ce-value, false),
            CHARS) :: SubDoCentre(L, 0)
SubDoCentre(itext (IMAP, CHARS) :: L, succ(COUNT))
        ⇐ itext(AddToMap(IMAP, ce-value, true),
            CHARS) :: SubDoCentre(L, COUNT)
SubDoCentre(cmd(C,ARG) :: L, COUNT)
        ⇐ cmd(br,null) :: SubDoCentre(L,
            NewValue(COUNT, ARG, succ(0),
            0, Huge)) if C = ce
        ⇐ cmd(C,ARG) :: SubDoCentre(L, COUNT)
            ifnot
```

The first equation for **SubDoCentre** corresponds to the trivial case when it has exhausted its input.

The next two correspond to the first element of the list of itext-or-command-lines being an itext-line. If the second argument, a count of the number of lines still to be centered, is 0, the centering value added into the infomap (done by **AddToMap**) is false, and **SubDoCentre** is applied to the remainder of the input. If it is nonzero, the centering value added is true, and the count is decremented in the recursive call of **SubDoCentre**.

The last equation corresponds to the first element of the list of itext-or-command-lines being a command-line. Different actions take place depending on whether or not the command is a ce command—if it is, then a br command-line is inserted, and **SubDoCentre** is applied to the remainder of the input with the count adjusted appropriately (calculated by **NewValue**); if it is not, then the command is passed through and **SubDoCentre** applied to the remainder of the input with its count unchanged.

An equation of the form

```
pattern ⇐ expression$_1$ if predicate$_1$
        ⇐ expression$_2$ if predicate$_2$
          . . .
        ⇐ expression$_n$ ifnot
```

rewrites to the first expression who's predicate following an if is true, and the expression before the ifnot if they are all false. This is useful here because it is impossible to write the pattern for a single equation to match the case of a cmd(C,ARG) command-line where C is anything but a ce command.

Portions from the LINES Stage of Processing

Within this stage text lines for output are formed. We show the functions that are used to produce unfilled lines.

Unfilled-I-Lines takes an infomap and a list of characters, and adds the infomap to the characters to form an unfilled itext-line. The desired indentation is obtained by prefixing the characters with blanks, equal in number to the ti-value (temporary indentation) in the infomap. If there are so many characters that, after indenting, they would extend beyond the right margin (again, a value in the infomap), then the excess characters are to be put into a successive line or lines. Hence the output of **Unfilled-I-Lines** is a list of itext-lines, rather than a single one. Subsidiary function **UnfilledLines** handles the details of splitting the characters over multiple lines if necessary.

```
Unfilled-I-Lines : infomap x list character → list
                    itext-or-command-line
Unfilled-I-Lines(MAP, CHARACTERS)
    ⇐ AddMap(MAP, UnfilledLines(MakeBlanks
        (INDENTATION)<>CHARACTERS,
        RIGHT-MARGIN))
      where INDENTATION = RetrieveFromMap
        (MAP, ti-value)
      where RIGHT-MARGIN = RetrieveFromMap
        (MAP, rm-value)

UnfilledLines : list character x num → list
                text-or-command line
UnfilledLines(CHARACTERS, RIGHT-MARGIN)
    ⇐ text(FIRST-LINE-CHARACTERS)
        :: UnfilledOverflowLines(REMAINDER)
    where FIRST-LINE-CHARACTERS, REMAINDER =
        SplitCharacters(RIGHT-MARGIN, CHARACTERS)
```

UnfilledLines produces a list of unfilled text-lines from the list of characters. The first text-line may have a width of at most right-margin. **SplitCharacters** splits off only as many characters as may be fit into this width, and the remainder are passed on to **UnfilledOverflowLines**, to put into unindented pagewidth wide lines.

```
UnfilledOverflowLines : list character → list
                        itext-or-command-line
UnfilledOverflowLines([ ]) ⇐ [ ]
UnfilledOverflowLines(CHAR :: CHARACTERS)
    ⇐ text(LEADING-CHARS) :: UnfilledOverflowLines
        (REMAINDER)
      where LEADING-CHARS, REMAINDER =
      SplitCharacters(PageWidth, CHAR :: CHARACTERS)
```

UnfilledOverflowLines produces lines of width at most **PageWidth** from the characters it is given. No indentation is performed on these lines.

MakeBlanks and **SplitCharacters** have relatively trivial definitions (omitted here).

For example,

characters = AVeryLongLineWithFarTooManyCharactersInIt.
indentation = 5
right margin = 12
page width = 14

should result in splitting the characters into

```
     AVeryLo
ngLineWithFarT
ooManyCharacte
rsInIt.
```

Note that it is easy to make minor modifications to the formatting of unfilled lines. For example, should we wish to simply discard the excess characters that do not fit onto the first line, we would modify **UnfilledOverflowLines** to always return [ ].

Alternatively, if we still wanted the excess characters put onto successive lines, but wanted those lines indented by the same amount as the first line, then we would modify **Unfilled-I-Lines** and **UnfilledLines** to pass the indentation value through to **UnfilledOverflowLines**, and modify the latter to append that many blanks to the list of characters given to **SplitCharacters**; thus,

```
UnfilledOverflowLines( CHAR :: CHARACTERS ,
                       INDENTATION )
    ⇐ text(LEADING-CHARS) :: UnfilledOverflowLines
         ( REMAINDER , INDENTATION )
       where LEADING-CHARS, REMAINDER =
         SplitCharacters(PageWidth , MakeBlanks
           (INDENTATION)<>CHAR :: CHARACTERS)
```

Acknowledgment

The author would like to thank all his colleagues at the Department of Artificial Intelligence, University of Edinburgh, for their help, in particular R. Burstall and J. Darlington for providing the original inspiration and continuing support and encouragement, and D. MacQueen for revealing discussions about specification.

References

[1] R. Aubin, "Mechanizing structural induction," Ph.D. dissertation, Univ. Edinburgh, Edinburgh, 1976.

[2] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 613-641, Aug. 1978.

[3] R. Balzer, N. Goldman, and D. Wile, "On the transformational implementation approach to programming," in *Proc. 2nd Int. Conf. Software Eng.*, Oct. 1976, pp. 337-344.

[4] F. L. Bauer, P. Partsch, P. Pepper, and H. Wossner, "Techniques for program development," in *Infotech State of the Art Report: Software Engineering Techniques*. Infotech Inform. Ltd, 1977, pp. 25-50.

[5] R. S. Boyer and J. S. Moore, "Proving theorems about LISP functions," *J. Ass. Comput. Mach.*, vol. 22, pp. 129-144, Jan. 1975.

[6] —, *A Computational Logic*. New York: Academic, 1979.

[7] W. H. Burge, *Recursive Programming Techniques*. Reading, MA: Addison-Wesley, 1975.

[8] R. M. Burstall, "Design considerations for a functional programming language. The software revolution," in *Proc. Infotech State of the Art Conf.*, Copenhagen, 1977, pp. 45-57.

[9] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *J. Ass. Comput. Mach.*, vol. 24, pp. 44-67, Jan. 1977.

[10] R. M. Burstall, D. B. MacQueen, and D. T. Sannella, "HOPE: An experimental applicative language," in *Proc. 1980 LISP Conf.*, Stanford, CA, 1980, pp. 136-143.

[11] K. L. Clark and F. G. McCabe, "The control facilities of IC-PROLOG," in *Expert Systems in the Microelectronics Age 1980, Proc. AISB Summer School on Expert Syst.*, July 1979, Edinburgh.

[12] O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare, *Structured Programming*. New York: Academic, 1972.

[13] J. Darlington, "Program transformation and synthesis: Present capabilities," Dep. Artificial Intell., Univ. Edinburgh, Tech. Rep. DAI43, 1977; appeared in *Artificial Intell.*, Mar. 1981.

[14] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica*, vol. 6, pp. 41-60, 1976.

[15] J. Darlington and M. S. Feather, "A transformational approach to program modification," Dep. Comput. Contr., Imperial College, London, Tech. Rep. 80/3, 1980.

[16] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[17] M. S. Feather, "A system for developing programs by transformation," Ph.D. dissertation, Dep. Artificial Intell., Univ. Edinburgh, Edinburgh, 1979.

[18] —, "A system for assisting program transformation," Dep. Artificial Intell., Univ. Edinburgh, Edinburgh, Tech. Rep. DAI124; see also *TOPLAS*, Jan. 1982.

[19] M.J.C. Gordon, A.J.R.G. Milner, L. Morris, M. Newey, and C. Wadsworth, "A metalanguage for interactive proof in LCF," in *Proc. 5th ACM POPL Symp.*, Tucson, AZ, 1978, pp. 119-130.

[20] B. W. Kernighan and P. J. Plauger, *Software Tools*. Reading, MA: Addison-Wesley, 1976.

[21] R. Kowalski, "Algorithm = logic + control," *Commun. Ass. Comput. Mach.*, vol. 22, pp. 424-436, July 1979.

[22] J. Schwarz, "Using annotations to make recursion equations behave," Dep. Artificial Intell., Univ. Edinburgh, Edinburgh, Tech. Rep. DAI43, 1977.

Martin S. Feather received the B.A. and M.A. degrees in mathematics and computer science from Cambridge University, England, in 1975 and 1976, respectively, and the Ph.D. degree in artificial intelligence from Edinburgh University, Edinburgh, Scotland, in 1979.

In October 1979 he joined the Information Sciences Institute, University of Southern California, Marina del Rey, CA, as a Research Associate on the Transformational Implementation Project at ISI, a project to develop the basis for a system which aids a user in transforming high-level abstract programs to very efficient, concrete programs. His professional interests center around program specification and development.